

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 1.1

## Table of Contents

[**CS 230 Project Software Design Template** 1](#_Toc115077317)

[**Table of Contents 2**](#_Toc115077318)

[**Document Revision History 2**](#_Toc115077319)

[**Executive Summary 3**](#_Toc115077320)

[**Requirements 3**](#_Toc115077321)

[**Design Constraints 3**](#_Toc115077322)

[**System Architecture View 3**](#_Toc115077323)

[**Domain Model 3**](#_Toc115077324)

[**Evaluation 4**](#_Toc115077325)

[**Recommendations 5**](#_Toc115077326)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 7/20/2025 | Matthew Scott | Initial Version |
| 1.1 | 8/1/2025 | Matthew Scott | Revised for accuracy and specificity |
| 1.2 | 8/17/2025 | Matthew Scott | Recommendations have been revised for accuracy, specificity and grounded context. |

**Instructions**

Fill in all bracketed information on page one (the cover page), in the Document Revision History table, and below each header. Under each header, remove the bracketed prompt and write your own paragraph response covering the indicated information.

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room has an Android game called Draw It or Lose It and wants a web‑based version that works on different devices. We will build a web application that uses a single GameService class to manage games, teams, and players. The GameService will be a singleton so only one instance runs at a time. We will check game and team names for uniqueness using simple loops before adding them. The classes will share common fields in an Entity base class and use straightforward OOP principles to keep the design clean and efficient.

## Requirements

The application must support one or more teams per game. Each team must have multiple players. Game and team names must be unique so users can see if a name is already in use. Only one instance of the GameService can run in memory at any time. Every game, team, and player must get a unique identifier. The application must run in a web browser and communicate with a backend API.

## [Design Constraints](#_2et92p0)

Singleton Service: We must use the singleton pattern for GameService so only one instance exists. This means we need a private constructor, a static instance field, and a public getInstance method.

Unique Name Checks: We must loop over existing games, teams, or players before adding new ones. That means each add operation has an extra search step, which can slow down adding if lists get long.

Web‑Based Distributed: The game runs in browsers on different platforms and talks to a backend service over HTTP. We must use RESTful APIs and handle network delays, errors, and security.

## [System Architecture View](#_ilbxbyevv6b6)

Please note: There is nothing required here for these projects, but this section serves as a reminder that describing the system and subsystem architecture present in the application, including physical components or tiers, may be required for other projects. A logical topology of the communication and storage aspects is also necessary to understand the overall architecture and should be provided.

## [Domain Model](#_8h2ehzxfam4o)

The UML class diagram shows an Entity base class with id and name fields. Game, Team, and Player all extend Entity, inheriting those fields and adding any specific behavior. The GameService class holds lists of Game, Team, and Player objects and provides methods to add and get them. Key OOP principles used:

* Inheritance: Entity centralizes common attributes so subclasses don’t repeat code.
* Encapsulation: Fields are private and accessed through methods to keep data safe.
* Singleton Pattern: GameService uses singleton so only one service runs.
* Iterator Pattern: add and get methods loop over lists to find or check items.These patterns ensure only one service instance, unique names and ids, and clear class structure.

**"The Gaming Room UML diagram. The top of the diagram is labeled as com dot gamingroom. Test boxes are placed in two layers. The first layer has three text boxes and the second layer has four of them. In the first layer, the 'ProgramDriver' textbox points to 'SingletonTester' textbox. The 'ProgramDriver' textbox contains the text 'asterisk main round brackets.' The 'SingletonTester' textbox contains the text 'asterisk testSingleton round brackets.' The arrow between these two text boxes are labeled 'open two angle brackets uses close two angle brackets'. In the second layer, there are 'GameService', 'Game', 'Team', and 'Player' text boxes. The 'GameService' textbox has texts arranged in two layers. The first layer contains games colon List open angle bracket Game close angle bracket, nextGamesId colon long, nextPlayer Id colon long, nextTeamId colon long, and service colon GameService. The second layer contains GameService round brackets, getinstance round brackets colon GameService, addGame open parenthesis name colon String close parenthesis colon Game, getGame open parenthesis id colon long close open parenthesis colon Game, getGame open open parenthesis name colon String close open parenthesis colon Game, getGameCount round brackets colon int, getNextPlayerID round brackets colon long, and getNextTeamId round brackets colon long. The 'GameService' box is connected with the 'Game' textbox with a line labeled 'zero dot dt dot asterisk'.  The 'Game' textbox also contains text in two layers. The first layers contains the text teams colon List open angle bracket Team close angle bracket. The second layer has Game open round bracket id colon long comma name colon String close parenthesis, addTeam open parenthesis name colon String close parenthesis Team, toString round brackets colon String. The 'Game' textbox is connected with the 'Team' textbox with a line labeled 'zero dot dt dot asterisk'. The 'Team' textbox also contains text in two layers. The first layers contains the text players colon List open angle bracket Player close angle bracket. The second layer has Team open parenthesis id colon long comma name colon String close parenthesis, addPlayer open parenthesis name colon String close parenthesis colon Player, and toString round brackets colon String. The 'Team' textbox is connected with the 'Player' textbox with a line labeled 'zero dot dt dot asterisk'. It contains the text Player open parenthesis id colon long comma name colon String close parenthesis and toString round brackets colon String. The 'Game', the 'Team, and the 'Player' boxes point to the 'Entity' textbox in first layer. The 'Entity' textbox contains text in two layers. The first layer has the text id colon long and name colon String. The second layer has Entity round brackets, Entity open parenthesis id colon long comma name colon String close parenthesis, getId round brackets colon long, getName round brackets colon String, toString round brackets colon String.**

## [Evaluation](#_2o15spng8stw)

Using your experience to evaluate the characteristics, advantages, and weaknesses of each operating platform (Linux, Mac, and Windows) as well as mobile devices, consider the requirements outlined below and articulate your findings for each. As you complete the table, keep in mind your client’s requirements and look at the situation holistically, as it all has to work together.

In each cell, remove the bracketed prompt and write your own paragraph response covering the indicated information.

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Mac can run a local web server easily for testing with tools, but it’s uncommon in production and hardware is pricey. Xcode is free, but macOS hardware can raise costs. Must ensure backend supports HTTPS, user sessions, and scale | Linux is free and popular for web servers, with strong community support and stability. Most large websites today run on Linux because it can handle high traffic with low resource usage and has built-in tools for automation. No OS licensing cost, good for budget hosting. Must ensure backend supports HTTPS, user sessions, and scale | Windows servers work well with IIS and .NET, and have a GUI for admins. However, licensing costs are high | Mobile devices have limited power and battery, so they are not suitable for hosting. They serve only as clients. We would need to manually test layout and responsiveness on different screen sizes like iPhones, Android tablets, and older smartphones to make sure everything displays correctly. Android Studio is free, but iOS requires a $99 Apple dev license per year. Must ensure backend supports HTTPS, user sessions, and scale |
| **Client Side** | To support Mac clients we test in Safari and Chrome. We need Mac hardware for testing and know-how on Mac browsers. Modern browsers mostly behave the same, but UI testing is still needed per OS. | Linux clients need testing in Firefox and Chrome. Tools and browsers are free and easy to install. Modern browsers mostly behave the same, but UI testing is still needed per OS. | Windows clients need testing in Edge and Chrome. Most users are on Windows so we must ensure compatibility. Modern browsers mostly behave the same, but UI testing is still needed per OS. | For mobile clients we need responsive UI or a small app, expertise in HTML5/CSS or frameworks like React Native, and time to test many devices. Needs responsive design for different screen sizes and touch input. |
| **Development Tools** | We can use Eclipse, IntelliJ, or VS Code on Mac. Java and web frameworks run well here. Could be used by designers or frontend devs building cross-platform features. | On Linux we use Eclipse, VS Code, or editors like Vim. Java or Python tools are free. Likely handled by a dedicated web/backend team familiar with terminal-based tools. | On Windows we use Visual Studio, Eclipse, or VS Code. The same languages and frameworks work. May require a separate frontend team using Visual Studio or other Windows IDEs. | For mobile web we can use React, Ionic, or PWAs. For native apps we use Android Studio or Xcode if on Mac. Mobile team may split between Android Studio (Android) and Xcode (iOS). |

## Recommendations

Analyze the characteristics of and techniques specific to various systems architectures and make a recommendation to The Gaming Room. Specifically, address the following:

1. **Operating Platform**:

Use **Linux** on the server, for example Ubuntu LTS. It is stable, well supported, and has no OS license fees. It runs common web stacks well. We can host it on any major cloud or on-prem. Keep it simple: Nginx as the front web server, Java 17 runtime for the app, and PostgreSQL for data.

1. **Operating Systems Architectures**:

We’ll break the system into three parts, each on its own Linux machine or virtual server:

* **Front end**: HTML, CSS, and JavaScript shipped from the server and cached by the browser or a CDN.
* **API**: a REST service written in Java. It exposes endpoints for games, teams, and players.
* **Data**: PostgreSQL for relational data. Store large images in object storage (for example S3 or a similar bucket) and serve them through a CDN.  
  Put a load balancer in front of the API. Put Nginx in front of static files. Keep logs central.

1. **Storage Management**:

Use PostgreSQL for core data. Add unique constraints on game and team names to enforce no duplicates. Create daily automated backups and keep at least 7 to 30 days. For the 200 images at ~8 MB each, plan for ~1.6 GB plus growth. Store the images in object storage with versioning and lifecycle rules. Add a CDN in front so downloads are fast everywhere. Keep small thumbnails for previews to save bandwidth. Track image metadata in the database (id, URL, checksum, size). This makes it easy to replace or roll back a bad file.

1. **Memory Management**:   
     
   Run the API in the JVM with a clear heap size. Use connection pooling so we do not waste memory on too many DB connections. Cache hot data with a small Redis cache, like active game state or recent lookups. For images, decode only what we need. Prefer WebP or optimized JPEGs to cut memory during render. Free references quickly by scoping objects to short-lived methods. Monitor with simple JVM metrics, GC logs, and a health endpoint.
2. **Distributed Systems and Networks**:

All clients talk to the API over HTTPS using JSON. If one API instance is busy or down, the load balancer sends traffic to another. Clients should retry on safe reads and timeouts. Use stateless API calls so any instance can handle any request. Send events like “round started” via WebSocket or Server-Sent Events for near-real-time play. Plan for failures: timeouts, backoff, and clear error messages. Keep a simple status/health endpoint so ops can see if each part is up.

1. **Security**:

Use HTTPS everywhere. Store passwords with a strong hash like bcrypt or Argon2. Do not store plain text. For login, start with Basic Auth in dev, then move to JWT or OAuth 2.0 in production. Validate and sanitize all inputs on the server. Use role-based access control so only admins can create or rename games and teams. Protect the database with a private network, least-privilege users, and regular patches. Rotate secrets. Turn on CORS only for allowed origins. Add rate limiting and audit logs for sensitive actions. Encrypt backups and the image bucket at rest.